CONTENTS

Decorator Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

class Decorator{
      public void decorate(Button b,Graphics g,int border){
            for(int x=1;x<=border;x++)
                  //drawRoundRect(int x, int y, int width, int height, int arcWidth, int
arcHeight)
                  g.drawRoundRect(b.getX()-x,b.getY()-x,b.getWidth()+x*2,b.getHeight()
+x*2,8,8);
      }
};

public class DecoratorExample extends Frame implements ActionListener{
      Button ellipticButton,applyButton;
      Decorator d;
      Label borderLabel;
      TextField borderSizeTextBox;
      int buttonPressed=0,border;

      public DecoratorExample(){ // Constructor function
            // Initialise all the variables and create the objects
            d=new Decorator();
            borderSizeTextBox=new TextField();
            borderLabel=new Label("Border");
            ellipticButton=new Button("Ellipctic");
            applyButton=new Button("Apply");
            setLayout(null);

            // Add all the items to the frame and set their positions
            add(borderLabel);
            add(borderSizeTextBox);
            add(ellipticButton);
            add(applyButton);
            borderLabel.setBounds(20,50,50,30);
            borderSizeTextBox.setBounds(80,50,50,30);
            applyButton.setBounds(140,50,50,30);
            ellipticButton.setBounds(100,100,80,30);
            applyButton.addActionListener(this);
      }
```

```
	public void paint(Graphics g){
		// If Apply button is pressed, draw the borders to the ellipctic button.
		if(buttonPressed==1){d.decorate(ellipticButton,g,border);}
	}

	public void actionPerformed(ActionEvent e){
		buttonPressed=1;
		border=Integer.parseInt(borderSizeTextBox.getText());
		repaint();
	}

	public static void main(String args[]){
		DecoratorExample myExample=new DecoratorExample();
		myExample.setVisible(true);
		myExample.setSize(800,550);
	}
};
```

Factory Method Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

abstract class Shape extends Frame{
      String name="";
      public abstract void display();
};

class Circle extends Shape{
      public Circle(){
            MyWindowAdapter myWindow=new MyWindowAdapter(this);
            addWindowListener(myWindow);
            setVisible(false);
      }

      public void display(){
            setSize(400,400);
            setVisible(true);
      }

      public void paint(Graphics g){
            // drawOval(int x, int y, int width, int height)
            g.drawOval(250,250,150,150);
      }
};

class Rectangle extends Shape{
      public Rectangle(){
            MyWindowAdapter myWindow=new MyWindowAdapter(this);
            addWindowListener(myWindow);
            setVisible(false);
      }

      public void display(){
            setSize(400,400);
            setVisible(true);
      }

      public void paint(Graphics g){
            // drawRect(int x, int y, int width, int height)
            g.drawRect(100,100,400,400);
```

```
	}
};

class Creator{
	public static Shape factoryMake(String n){
		if(n.equals("Rectangle"))
			return new Rectangle();
		else
			return new Circle();
	}
};

class MyWindowAdapter extends WindowAdapter{
	Shape s;
	public MyWindowAdapter(Shape s){
		this.s=s;
	}

	public void windowClosing(WindowEvent we){
		s.setVisible(false);
	}
};

class MyWinAdapter extends WindowAdapter{
	FactoryExample s;
	public MyWinAdapter(FactoryExample s){
		this.s=s;
	}

	public void windowClosing(WindowEvent we){
		s.setVisible(false);
	}
};

class FactoryExample extends Frame implements ItemListener{
	Choice shapeChoice=new Choice();
	public FactoryExample(){
		MyWinAdapter myWindow=new MyWinAdapter(this);
		addWindowListener(myWindow);
		shapeChoice.addItem("Rectangle");
		shapeChoice.addItem("Circle");
		setLayout(new FlowLayout());
		add(shapeChoice);
		shapeChoice.addItemListener(this);
```

```
	}

	public void itemStateChanged(ItemEvent e){
		Shape s;
		s=(Shape)Creator.factoryMake(shapeChoice.getSelectedItem()); // "Rectangle"
or "Circle"
		s.display();
	}

	public static void main(String str[]){
		FactoryExample myFactoryFrame=new FactoryExample();
		myFactoryFrame.setVisible(true);
		myFactoryFrame.setSize(800,600);
	}
};
```

List Iterator Pattern

```
class Node{
      int value;
      Node next;
      Node previous;

      Node(int v){ // Create a new node
            value=v;
            next=null;
            previous=null;
      }
};

class ListIterator{
      ListIteratorExample myList;
      Node temp;

      ListIterator(ListIteratorExample passedList){
            this.myList=passedList;
      }

      int first(){
            temp=myList.start;
            return temp.value;
      }

      int next(){
            temp=temp.next;
            return temp.value;
      }

      int previous(){
            temp=temp.previous;
            return temp.value;
      }

      int current(){
            return temp.value;
      }
};

class ListIteratorExample{
      Node start; // the first starting node
```

```
	ListIteratorExample(){ // Constructor

		Node temp=null;
		// Let's create a new node and initialise this first node to 0
		Node newNode=new Node(0);
		newNode.previous=temp;
		newNode.next=null;
		temp=newNode;
		start=newNode;

		// Let's insert a few more nodes
		for(int i=1;i<10;i++){
			Node n=new Node(i);
			n.previous=temp;
			n.next=null;
			temp.next=n;
			temp=n;
		}
	}

	public static void main(String args[]){
		ListIteratorExample myList=new ListIteratorExample();
		ListIterator myIterator=new ListIterator(myList);

		System.out.println("The first node is : "+ myIterator.first());

		System.out.print("Iterating forward five spaces : ");
		for(int i=0;i<5;i++){
			System.out.print(myIterator.next()+" ");
		}

		System.out.println(" | Current node is "+myIterator.current()); // new line

		System.out.print("Iterating backward three spaces : ");
		for(int i=0;i<3;i++){
			System.out.print(myIterator.previous()+" ");
		}

		System.out.println(" | Current node is "+myIterator.current()); // new line
	}
};
```

Model View Controller Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

class view extends Frame{
	int points[];

	public view(){
		setVisible(true);
		setSize(800,550);
	}

	public void draw(int n[]){
		points=new int[n.length];
		for(int x=0;x<n.length;x++)
			points[x]=n[x];
		repaint();
	}

	public void paint(Graphics g){
		for(int x=0;x<points.length;x++)
			g.drawRect(200+(x*40),400-points[x],40,points[x]);
	}
};

class controller{
	static view myView;

	public controller(){
		myView=new view();
	}

	public void check(TextField t[]){
		int a[]=new int[t.length];
		for(int x=0;x<t.length;x++)
			a[x]=Integer.parseInt(t[x].getText());
		myView.draw(a);
	}
};

class MVCExample extends Frame implements TextListener{ // this is the model class
	controller c;
```

```
	TextField t[];

	public MVCExample(){
		c=new controller();
		// Let's have 12 text fields
		t=new TextField[12];
		// Setting the layout to Flow Layout so the boxes are arranged tabularly
		setLayout(new FlowLayout());
		// Adding the 12 text fields to the view, with default value set to 0
		for(int x=0;x<12;x++){
			t[x]=new TextField("0");
			add(t[x]);
			t[x].addTextListener(this);
		}
	}

	public void textValueChanged(TextEvent e){
		c.check(t);
	}

	public static void main(String args[]){
		MVCExample myModel=new MVCExample();
		myModel.setVisible(true);
		myModel.setSize(100,200);
	}
};
```

Publish Subscribe Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

class Subscribe extends Frame{
      List myList;

      public Subscribe(){
            setVisible(true);
            setSize(200,200);
            setLayout(null);
            myList=new List();
            myList.setBounds(0,20,200,200);
            add(myList);
      }

      public void inform(String str){
            myList.add("Notified Value : "+str);
      }
};

public class PublishSubscribeExample extends Frame implements ActionListener{ // this is
the publish class
      TextField numberTextField,dataTextField;
      Label numberLabel,dataLabel;
      Button createButton,notifyButton;
      int subs;
      Subscribe s[]=new Subscribe[10];

      public PublishSubscribeExample(){
            numberTextField=new TextField();
            dataTextField=new TextField();

            numberLabel=new Label("No of Objects");
            dataLabel=new Label("Data");

            notifyButton=new Button("Notify");
            createButton=new Button("Create");

            setLayout(null);

            numberLabel.setBounds(200,50,100,30);
```

```
            numberTextField.setBounds(300,50,100,30);
            createButton.setBounds(400,50,100,30);
            dataLabel.setBounds(200,90,100,30);
            dataTextField.setBounds(300,90,100,30);
            notifyButton.setBounds(400,90,100,30);

            add(numberLabel);
            add(numberTextField);
            add(createButton);

            createButton.addActionListener(this);
            notifyButton.addActionListener(this);
      }

      public void actionPerformed(ActionEvent e){
            if(e.getSource()==createButton){ // if create button is pressed then do this
                  subs=Integer.parseInt(numberTextField.getText());
                  for(int x=0;x<subs;x++)
                        s[x]=new Subscribe();
                  remove(numberLabel);

                  add(dataLabel);
                  add(dataTextField);
                  add(notifyButton);
            }

            if(e.getSource()==notifyButton){ // if notify button is pressed then do this
                  for(int x=0;x<subs;x++)
                        s[x].inform(dataTextField.getText());
            }
      }

      public static void main(String args[]){
            PublishSubscribeExample myPublisher=new PublishSubscribeExample();
            myPublisher.setVisible(true);
            myPublisher.setSize(800,550);
      }

};
```

Proxy Pattern

```
// Let's import the libraries we need
import java.awt.*;

class Proxy{
      public void initialise(int left,int top,int width,int height,Graphics g){
            g.drawRect(left,top,width,height);
            g.setColor(Color.black);
            g.fillRect(left,top,width,height);
      }

      public void draw(String text,int left,int top,Graphics g,ProxyExample
myProxyExample){
            g.setColor(Color.red);
            g.drawString(text,left,top);
      }
};

class ProxyExample extends Frame{
      Proxy myProxy;

      public ProxyExample(){
            setVisible(true);
            setSize(800,550);
            myProxy=new Proxy();
      }

      public void paint(Graphics g){
            myProxy.initialise(100,100,300,50,g);
            myProxy.draw("This is a test string",110,115,g,this);
            myProxy.draw("This is another test string",130,140,g,this);
      }

      public static void main(String args[]){
            new ProxyExample();
      }
};
```

Singleton Pattern

```
class Singleton{
	private static Singleton mySingleInstance;
	public int value;

	private Singleton(){} // "private" constructor is inaccessible to the outside world

	public static Singleton instance(){
		if(mySingleInstance==null){
			mySingleInstance=new Singleton();
			return mySingleInstance;
		}
		else{
			return mySingleInstance;
		}
	}
};

public class SingletonExample{
	public static void main(String args[]){
		Singleton myFirstInstance,mySecondInstance;
		myFirstInstance=Singleton.instance();
		mySecondInstance=Singleton.instance();

		myFirstInstance.value=5;
		System.out.println("The values after chaning the value of the first instance
are : ");
		System.out.println("myFirstInstance value is "+myFirstInstance.value+" and
mySecondInstanceValue is "+mySecondInstance.value);

		mySecondInstance.value=8;
		System.out.println("The values after chaning the value of the second instance
are : ");
		System.out.println("myFirstInstance value is "+myFirstInstance.value+" and
mySecondInstanceValue is "+mySecondInstance.value);
	}
};
```

Strategy Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

abstract class Scheduler{
      public abstract String schedule();
};

class Lifo extends Scheduler{
      String data="";
      public String schedule(){
            data="Lifo";
            return data;
      }
};

class Fifo extends Scheduler{
      String data="";
      public String schedule(){
            data="Fifo";
            return data;
      }
};

class Composition extends Frame{
      Scheduler myScheduler;
      String data="";
      public Composition(String n){
            MyWindowAdapter myWindow=new MyWindowAdapter(this);
            addWindowListener(myWindow);
            setVisible(true);
            setSize(200,200);
            if(n.equals("Lifo")){
                  myScheduler=new Lifo();
                  data=myScheduler.schedule();
            }
            else{
                  myScheduler=new Fifo();
                  data=myScheduler.schedule();
            }
      }
```

```
	public void paint(Graphics g){
		g.drawString(data,100,100);
	}
};

class MyWindowAdapter extends WindowAdapter{
	Composition myComposition;
	public MyWindowAdapter(Composition passedComposition){
		this.myComposition=passedComposition;
	}
};

class MyWinAdapter extends WindowAdapter{
	StrategyExample myStrategy;
	public MyWinAdapter(StrategyExample passedStrategy){
		this.myStrategy=passedStrategy;
	}
};

public class StrategyExample extends Frame implements ItemListener{
	Choice myChoice=new Choice();
	public StrategyExample(){
		MyWinAdapter myWindow=new MyWinAdapter(this);
		addWindowListener(myWindow);
		myChoice.addItem("Lifo");
		myChoice.addItem("Fifo");
		setLayout(new FlowLayout());
		add(myChoice);
		myChoice.addItemListener(this);
	}

	public void itemStateChanged(ItemEvent e){
		Composition myComposition=new Composition(myChoice.getSelectedItem());
	}

	public static void main(String args[]){
		StrategyExample myStrategy=new StrategyExample();
		myStrategy.setVisible(true);
		myStrategy.setSize(800,600);
	}
};
```

Template Pattern

```
// Let's import the libraries we need
import java.awt.*;
import java.awt.event.*;

abstract class Area extends Frame{
      String aboutMe="",myResult="";
      Area(){
            setVisible(true);
            setSize(400,400);
            MyWindowAdapter myWindow=new MyWindowAdapter(this);
            addWindowListener(myWindow);
      }

      public void templateMethod(){
            aboutMe="I calculate the areas and perimeters of figures.";
            myResult="The area is "+area()+" and the perimeter is "+perimeter();
      }

      public abstract double area();
      public abstract double perimeter();

      public void paint(Graphics g){
            g.drawString(aboutMe,50,100);
            g.drawString(myResult,50,200);
      }
};

class Rectangle extends Area{
      double length=21;
      double breadth=4;
      public double area(){return (length*breadth);}
      public double perimeter(){return 2*(length+breadth);}
};

class Circle extends Area{
      double radius=21;
      public double area(){return (3.14*radius*radius);} // Pie R-square
      public double perimeter(){return (2*3.14*radius);} // 2 Pie R
};

class MyWindowAdapter extends WindowAdapter{
      Area ofMyFigure;
```

```
	public MyWindowAdapter(Area ofPassedFigure){
		this.ofMyFigure=ofPassedFigure;
	}
};

class MyWinAdapter extends WindowAdapter{
	TemplateExample myTemplate;
	public MyWinAdapter(TemplateExample passedTemplate){
		this.myTemplate=passedTemplate;
	}
};

public class TemplateExample extends Frame implements ItemListener{
	Choice myChoice=new Choice();
	Area area;

	public TemplateExample(){
		MyWinAdapter myWindow=new MyWinAdapter(this);
		myChoice.addItem("Rectangle");
		myChoice.addItem("Circle");
		setLayout(new FlowLayout());
		add(myChoice);
		myChoice.addItemListener(this);
	}

	public void itemStateChanged(ItemEvent e){
		if(myChoice.getSelectedItem()=="Rectangle")
			area=new Rectangle();
		else
			area=new Circle();
		area.templateMethod();
	}

	public static void main(String args[]){
		TemplateExample myTemplate=new TemplateExample();
		myTemplate.setVisible(true);
		myTemplate.setSize(800,600);
	}
};
```

Layered Architecture in Windows 2000

Windows 2000 is built upon a layered approach, similar to the UNIX operating system. One advantage of the layered operating structure is that each layer of code is given access only to the layer below it (interfaces and data structures). This structure also allows the operating system to be debugged, starting at the lowest layer and adding one layer at a time until the whole system works correctly. Layering also makes it easier to enhance the operating system; one entire layer can be replaced without affecting other parts of the operating system.

Windows 2000 is a portable operating system because of two design decisions. First, the operating system was written in ANSI C, a language that enables programs to be ported easily to other hardware architectures. Second, all parts of Windows 2000 that must be written for a specific hardware are isolated in an area called the Hardware Abstraction Layer (HAL). To move Windows 2000 to a new hardware platform, developers need to do little more than recompile the C code for the new hardware and create a new HAL. Designing an OS around a HAL means that a large portion of the code is exactly the same between hardware platforms. This also means that only the small slice of code that interfaces with the computer's hardware needs to be rewritten as Windows 2000 is ported between different processor architectures. Thus, it provides a high level of portability.

The two modes that Windows 2000 operates in are kernel mode and user mode

Kernel Mode

In this mode, the software is able to access the hardware and system data, as well as access all other system resources. The kernel mode has the following components:

- **Executive**. Contains components that implement memory management, process and thread management, security, I/O, interprocess communication, and other base operating system services. For the most part, these components interact with one another in a modular, layered fashion.
- **Microkernel**. The Microkernel's primary functions are to provide multiprocessor synchronization, thread and interrupt scheduling and dispatching, and trap handling and exception dispatching. During system startup, it extracts information from the Registry, such as which device drivers to load and in what order.
- **Hardware Abstraction Layer (HAL)**. The HAL is the code associated with Windows 2000 that changes with the hardware the operating system is being run on. Thus, it becomes compatible with multiple processor platforms. The HAL manipulates the hardware directly.
- **Device drivers**. Device drivers send and receive load parameters and configuration data from the Registry.
- **Windowing and graphics system**. This system implements the graphical user interface (GUI).

User Mode

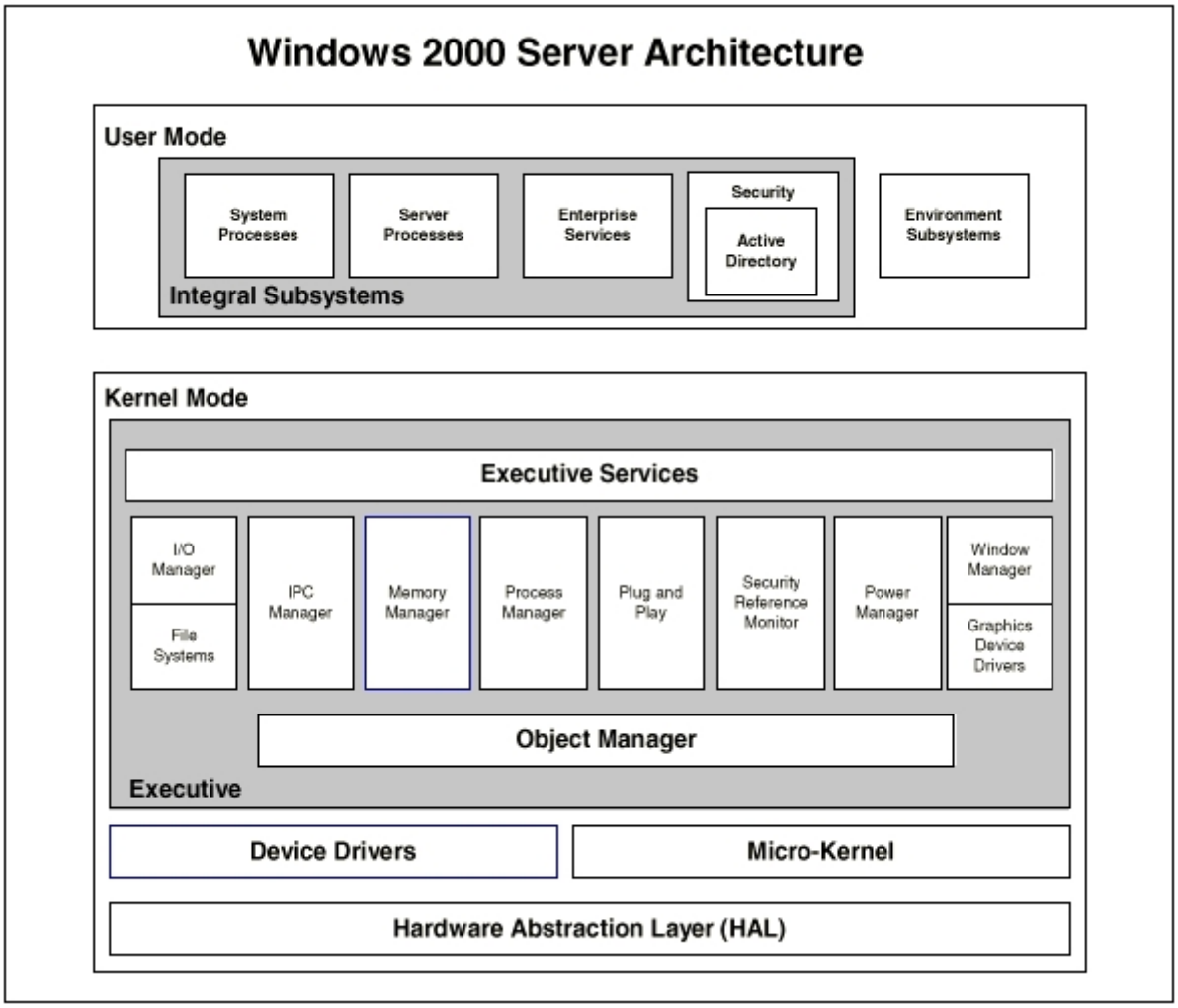
Software in the user mode cannot access hardware directly. The user mode-protected subsystem has four primary responsibilities:

- Special system support processes, such as the logon process and the session manager.
- Windows 2000 services that are server processes, such as the Event Log and

Schedule services.

- Environment subsystems that provide an operating system environment by exposing the native operating system services to user applications. They include Win32, POSIX, and OS/2 subsystems.
- User applications—either Win32, Windows 3.1, MS-DOS, POSIX, or OS/2.

User applications do not call the native Windows 2000 operating system services directly; instead, they go through subsystem dynamic link libraries (DLLs). The subsystem dynamic link libraries translate a documented function into the appropriate undocumented Windows 2000 system service calls. The protected subsystems are divided into two groups—environment subsystem and integral subsystem—which are described in the following sections.



Layered Architecture in Networking

layer 7 application	Applications and application interfaces for OSI networks. Provides access to lower layer functions and services.
layer 6 presentation	Negotiates syntactic representations and performs data transformations, e.g. compression and code conversion.
layer 5 session	Coordinates connection and interaction between applications, established dialog, manages and synchronizes data flow direction.
layer 4 transport	Ensures end-to-end data transfer and integrity across the network. Assembles packets for routing by Layer 3.
layer 3 network	Routes and relays data units across a network of nodes. Manages flow control and call establishment procedures.
layer 2 data link	Transfers data units from one network unit to another over transmission circuit. Ensures data integrity between nodes.
layer 1 physical	Delimits and encodes the bits onto the physical medium. Defines electrical, mechanical and procedural formats.

Design Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns can be broadly divided into (a) Creational Patterns (b) Structural Patterns and (c) Behavioral Patterns.

Creational patterns

Abstract Factory : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method : Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Builder Pattern : Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Prototype Pattern : Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton Pattern : Ensure a class only has one instance, and provide a global point of access to it.

Structural patterns

Adapter Pattern : Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge Pattern : Decouple an abstraction from its implementation so that the two can vary independently.

Composite Pattern : Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator Pattern : Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade Pattern : Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight Pattern : Use sharing to support large numbers of fine-grained objects efficiently.

Proxy Pattern : Provide a surrogate or placeholder for another object to control access to it.

Behavioral patterns

Chain of responsibility Pattern : Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command Pattern : Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter Pattern : Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator Pattern : Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator Pattern : Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento Pattern : Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer Pattern : Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State Pattern : Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy Pattern : Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Pattern : Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor Pattern : Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.